Lecture 13: Overview

- NP-Completeness (Chapter 13)
NP Completeness

• Almost all algorithms we have covered so far have been polynomial-time algorithms
• Worse-case running time is $O(n^k)$ for some constant $k$
• Are all problems solvable in polynomial time?
Halting Problem

• Halting problem:
  • Can we write an algorithm that takes in a program and always determine whether the program will halt (i.e. complete its execution) or not?
  • The answer is no
  • Why not?
Halting Problem

• Suppose that we had such an algorithm A
• Consider the program P:
  • Run A on itself (P)
  • If A says we halt, loop
  • If A say we loop, halt
• Halting problem can’t be solved
• Undecidable class
Categorizing Problems

- Like to categorize the difficulty of problems
- Difficulty is that different machine models may have change the complexity of the program
- Consider a machine that stores everything on a tape...memory accesses aren’t O(1) anymore
- Turns out that all Turing machines are equivalent within a polynomial factor
Class P

- Create the first class of problems P (Polynomial)
- These problems can be solved in polynomial time
Class NP

- Non-deterministic polynomial
- Class of problems that we can verify the solution of in polynomial time
Big Open Problem in Computer Science

• Does $P=NP$?
• If so, then a bunch of hard problems (factoring, etc) are actually easy.
  • Seems unlikely
  • Lots of people have been looking for good solutions to these problems
• But so far there is no proof that $P \neq NP$
Decision vs. Optimization

- Talking about decision problems (is there a solution)
- Not talking about optimization problems (what is the best solution)
- Can convert optimization problems into decision problems (shortest path => is there a path of less than length l for some l)
NP-complete

- NP-complete problems are at least as hard as any other problem in NP
- How to prove:
  - Need to show that we can reduce any other problem in NP to this problem in polynomial time
  - Show that we can reduce another NP-complete problem to this problem in polynomial time
First NP-Complete Problem

- Circuit Satisfiability
- Given a logic circuit of AND, OR, and NOT gates, is there an input that causes the output to be true
Formalization

- Need to encode problem instance into a binary string using some type of encoding.
- Algorithm that “solves” a problem actually takes an encoding of a problem instance as input.
- We state that an algorithm solves a concrete problem in $O(T(n))$ if when it is provided an instance $i$ of length $n = |i|$, the algorithm can produce a solution in $O(T(n))$ time.
- An algorithm is polynomial time solvable if there exists an algorithm to solve in $O(n^k)$ time.
Encoding Matters

• Suppose we have an algorithm that operates on an integer k specified as k 1’s and is \( \Theta(k) \)
• With a normal encoding, \( n = \log|k| \) and the algorithm is \( O(2^n) \)
• Normally rule out really bad encodings of the problem
Polynomial-Time Computable

• Say that a function \( f: \{0,1\}^* \rightarrow \{0,1\}^* \) is polynomial computable is there exists a polynomial time algorithm that given an input \( x \) in \( \{0,1\}^* \) produces \( f(x) \).

• Two encodings \( e_1 \) and \( e_2 \) are polynomial related if there exists two polynomial-time computable functions \( f_{12} \) and \( f_{21} \) such that \( f_{12}(e_1(i)) = e_2(i) \) and \( f_{21}(e_2(i)) = e_1(i) \).
Hamiltonian Cycles

• Given graph $G=(V,E)$
• Find simple cycle in graph that contains each vertex $V$
• Hamiltonian-cycle problem:
  Does graph $G$ have a hamiltonian cycle?
• $\text{HAM-CYCLE} = \{ <G> : G \text{ is a hamiltonian graph} \}$
How to Decide?

- One solution: check each possible permutation of the vertices to see if it is a Hamiltonian path.
- Encoding: For adjacency matrix, number of vertices $= \Omega(\sqrt{n})$ where $n = |<G>|$.
- $m!$ possible permutations.
- Runtime is $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$.
Verification

- If we are given a sequence of vertices, we can verify whether it is a hamiltonian cycle easily in polynomial time.
- Idea: Algorithm takes in
  - Input string $x$
  - Certification of solution $y$
Complexity Class NP

- Class of problems that can be verified by a polynomial-time algorithm
Reducibility

• How do we prove a problem NP-complete?
• Intuition: problem Q can be reduced to Q’ if any instance of Q is “easily rephrased” into an instance of Q’
• We say that L₁ is polynomial-time reducible to L₂ if there exists a polynomial time computable function f: \( \{0,1\}^* \rightarrow \{0,1\}^* \) that converts an instance of one problem into an instance of the second problem

Call f the reduction function and a polynomial-time algorithm F that computes f is called a reduction algorithm
A problem is NP-complete if

1. It is in NP
2. Instances of all other NP problems can be converted to an instance of this problem in polynomial time (NP-hard)

#2 tells us that this problem is at least as hard as any other in NP
Can reduce all of NP to Circuit Satisfiability

• Idea:
  • Represent computation of A as sequence of configurations
  • Each configuration includes PC, machine state, input, certificate, working storage
  • Use combinatorial circuit M that implements computer hardware
Reduction

- Reduction algorithm uses bound to compute number of steps $T(n)$ that algorithm A takes for problem
- Generates $T(n)$ copies of $M$ which feed the configuration to the copy of $M$
- Run circuit satisfiability on this converted problem
- Need to prove that reduction is polynomial